

Software Security & Secure Programming

Lab session 1 : language security weaknesses

Before you start :

- You can work either on your own laptop or on an Ensimag (Ubuntu) machine ...
- Copy the tar file
<https://im2ag-moodle.univ-grenoble-alpes.fr/mod/resource/view.php?id=31988>
and untar it into a directory of your choice ...
- There are numerous exercices, you are expected to do **at least 2 exercises from Part 1 and one exercise from Part 2**, and to write a **report** including **both** :
 - your **detailed answers** for these **3 exercices** ;
 - a **short explanation** about the problem raised and the solution proposed for one of the sub-rules of rules 04, 06, 08 or 12 from the web site
<https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>
- This report should be **uploaded** on the Moodle course page¹ before **October the 21st**. It **should be** written by **group of 2 students** ("binôme").
- Parts of this lab consist in discovering security features (vulnerabilities, protections, tools) which may have not yet been seen during the course. This is a deliberate choice, we will come back to that in the next lectures ...

1. <https://im2ag-moodle.univ-grenoble-alpes.fr/mod/assign/view.php?id=36469>

Part 1

Exercise 1 Optimise (arithmetic overflow)

Look at the source code of `optimise.c`.

1. Compile it, execute it and explain the result obtained in each following case :

no option :

```
gcc -o optimise1 optimise.c
```

optimisation option :

```
gcc -O2 -o optimise2 optimise.c2
```

overflow detection option :

```
gcc -fno-strict-overflow -o optimise3 optimise.c
```

optimisation and overflow detection option :

```
gcc -O2 -fno-strict-overflow -o optimise4 optimise.c
```

(Look at the `gcc` manual to know the meaning of `-O2` and `-fno-strict-overflow3` ...).

2. Propose a solution to make this code *secure* whatever are the options used to compile it. You can use the following rule : <https://www.securecoding.cert.org/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>

Exercise 2 WinLoose (buffer overflow)

Look at the source code of the C program `winloose.c`. This program takes as input two integer arguments on the command line (`argv[1]` and `argv[2]`).

1. Compile this program with `gcc` using the following command :

```
gcc -fno-stack-protector -o winloose winloose.c
```

(Have a look at the `gcc` manual to know the meaning of `-fno-stack-protector` ...).

Execute it with some random arguments :

```
./winloose 5 10
./winloose 2 17
etc.
```

This program may lead to several possible results :

- print "You loose"
- infinite loop
- crash
- etc.

2. Explain each different result you get, "guessing" the execution stack layout. In particular try to find a program input allowing to print "You win"!

2. if you see the same behavior than in the previous case you should add `-fstrict-overflow` to this command
3. and `-fstrict-overflow` in case you used it

3. You can confirm and/or complete your results for the previous question by retrieving the actual stack layout from the executable code produced by the compiler. This can be done in several ways, e.g. :
 - using a *decompiler*, like **Ghidra** (available on the Ensimag machine, instructions are available here)
 - using a *disassembler*, like **objdump**
 - using a *debugger*, like **gdb**
- 4.
5. Compile now the C program **winloose.c** with and without the “stack protection” enabled :


```
gcc -fstack-protector -o winloose-protected winloose.c
gcc -fno-stack-protector -o winloose-unprotected winloose.c
```

What do you obtain now when running these executable codes with the inputs you provided for question 1 ?

6. Disassemble or decompile the protected program using **objdump** or **Ghidra**.

Look at the assembly (or decompiled) code of function **<main>** to retrieve how the stack protection mechanism is implemented.

Exercise 3 ExecShell (input sanitization, use-after-free)

Look at the source code of the C program **exec_shell.c**. This program takes as input one directory name and prints its content (like the **ls** command).

Compile this program with **gcc** : **gcc -o exec_shell exec_shell.c**

Run it : **./exec_shell /tmp**

If the argument string is too long, then an error message is printed and the user is requested to enter a character string.

1. Explain why this program is *vulnerable*.
2. Find how you can use this program to execute any shell command of your choice (e.g, **/bin/sh**, **xcalc**, etc.) using the two following attack patterns :
 1. shell command injection
 2. use-after-free vulnerability

Exercise 4 Path traversal (input sanitization)

The file **path_traversal.java** contains a Java program allowing to print the content of a user profil file on a (fictive) Linux filesystem those root is **myroot**.

Compile this program (**javac path_traversal.java**)

We assume the following *attacker model* :

- `myroot` is **remote** and can only be accessed by users through this Java program ;
- the source code of `path_traversal.java` is not available to users, and it cannot be easily reversed.

1. As an “attacker”, run `java path_traversal`
 - first, without argument ;
 - then, with a random argument (for instance your login name).

What do you learn ?

2. According to the structure of a Linux filesystem you know that the “password file” is stored in `/etc/passwd`. How can you use `path_traversal.class` to print the content of this file ?

Have a look at <https://cwe.mitre.org/data/definitions/34.html> ...

3. As a developper, how could you improve the code of `path_traversal.java` in order to prevent this attack ? Several solutions can be considered :
 - define bounds for expected string (like OVH and free for accounts’ password on their manager) so the number of check to do on the input string could be easily defined (have a look at filesystems constants for this)
 - (properly) sanitize input string
 - have a look at
 - http://en.wikipedia.org/wiki/Directory_traversal_attack
 - <http://cwe.mitre.org/data/definitions/22.html>

Exercise 5 A DoS attack ! (input sanitization)

The (simple !) program `integers-sum.c` expects a sequence of integer values from the user (ending by `-1`) and prints the sum of the received input (expected `-1`). However, there is a (simple !) way - as a regular user - to make it entering an **infinite loop**, leading to Denial-of-Service (DoS) attack.

1. compile and run this code with regular input sequences to check that it works well under nominal conditions.
2. find the way to trigger this attack (**clue** : you simply need to enter some simple unexpected input, using a very basic *fuzzing* technique).
3. explain the vulnerability, and where it does come from.
4. propose a more secure version of this code, preventing the former attack.

Exercise 6 Exploiting a Use-After-Free (use-after-free)

The objective of this exercise is to show how a use-after-free vulnerability can be exploited by an attacker to get an *arbitrary code execution*. The commands to be executed in the following questions can be copy-pasted from the file `exploit-uaf2.txt`.

1. Have a look at the file `uaf2.c` to spot the *use-after-free*.
Compile this file using option `-z execstack` to set the stack as “executable” (which is not

a default option).

```
gcc -z execstack -o uaf2 uaf2.c
```

2. What do you obtain when executing the following command : ./uaf2 foo
Explain why you get this result ...
3. Execution of **uaf2** can be hijacked by an attacker and may lead to an arbitrary code execution by giving as input a sequence of processor instructions (called a *shellcode*). An example of such a shellcode allowing to **open a shell** under Linux x86/64 is given for instance here : <https://www.exploit-db.com/exploits/46907>.

Run **uaf2** with this shellcode a command line argument (see file **exploit-uaf2.txt**).

4. Re-compile now your code using Adress-Sanitizer in order to enforce memory safety :

```
gcc -g -fsanitize=address -z execstack -o uaf2 uaf2.c
```

Check that the previous “exploit” does not work anymore ...

Part 2

See this file ...